RedEye Advanced Programming Manual

for software version 2.5.0

Contents

Part I: Overview **1**

Introduction **3**
Technical Support Contacts *3*

RedEye Architecture **4**
True Client-Server Design 4
Relational Database 6
Web Interface 7
Scripting Engine 9

Part II: Scripting Basics **11**

Introduction to Lua **13**
Basic Syntax 14
Variables and Scope 16
Operators and Keywords 17
Conditional Statements *19*
Conditional Loops *19*
Iterative Loops *20*
Function Declarations *22*
Referencing Other Libraries 24
Strings 25
String Casing *28*
Converting Between ASCII and Byte Values *28*
Substrings *29*
Pattern Matching *30*

Scripting Basics **34**
Two Kinds of Scripts 34
Command Scripts *34*
Action Scripts *34*
Impact on Scripting *35*
A Simple Launch Activity Script 36
Debugging the Launch Activity Script 40

The PortListener Framework **42**
Behind the Curtain 42
Message Queuing *44*
Inbound Data Processing *45*
Customizing Your PortListener 46
A Sensor PortListener *47*
A Serial PortListener *50*

Command Scripts **54**

Other Scripting Tools 57
File System Access 57
Proper Use of the Flash File System 57
Accessing Devices Directly 58
Multithreading 59
XML Processing 59
SQLite Databases 59

Part I: Overview

Introduction

This manual is a companion to the *RedEye User Manual*, specifically focused on the advanced customization options available beginning with the release of RedEye app v2.5.0. Customers looking for basic usage information and general reference on the RedEye application should refer to the *RedEye User Manual*.

Although designed to be intuitive and simple to use, the topics covered in this manual are by nature more complex than other aspects of RedEye configuration and use. The flexibility which some of these features provide is invaluable in creating a fully-integrated and robust system, but that flexibility can also lead to problems if not approached properly. As a result, we recommend that customers first familiarize themselves with basic RedEye operation before approaching the advanced customization topics discussed here. Before jumping into advanced configuration, we recommend creating a backup of your RedEye configuration so that you can “reset” back to a known good state.

Of course if you do find yourself stuck, please make use of our many technical support options, including our technical support line and online customer support forums. As always, we are here to help.

Technical Support Contacts

By phone: 617-299-2000, option 1
By email: support@thinkflood.com
Online: http://thinkflood.com/support/redeye/

RedEye Architecture

From the very beginning we designed RedEye to be more than a simple remote control. Remote controls – even “universal” ones – have come to take on a rather specific meaning. Typically infrared-based, these wand-style controllers come covered in buttons and have a single function: to control specific pieces of equipment. By contrast, in RedEye we were taking something more generic – a smartphone – and repurposing it to function also as a remote control.

This repurposing of the smartphone comes with both advantages and disadvantages. On the negative side, smartphone batteries do not last as long as dedicated remote control batteries, and therefore the smartphone needs to be turned off periodically to conserve power. Also, the multi-purpose nature of the smartphone means that the remote control function exists as only one application among many, with the implication that there would be additional latency in launching the application after using the phone for some other purpose. There were other disadvantages, too – the lack of dedicated, physical buttons being one that often comes up in discussion.

To make RedEye a useful product, its advantages needed to overcome these disadvantages. Some of these advantages are easy to tease out. For example, the touchscreen interface of today’s smartphones provides nearly infinite customization potential, so that was a no-brainer. But some of these advantages are not so evident, and can quickly become impossible depending on certain early design choices.

True Client-Server Design

One of our early realizations was that while using a single smartphone as a remote control introduces several disadvantages, the ability to use many smartphones – or, taken a step further, just about any networked computer – as control interfaces provides major opportunities. At its essence, RedEye is not about converting an iPhone into a remote control. Instead, it is about bringing all kinds of previously isolated devices onto the Internet. In other words, RedEye becomes the network interface for TVs, A/V receivers, DVD players, air conditioners, and a host of other products. In theory, these devices are now available wherever you get an Internet connection, and you can control them with whatever networked computer you have handy, whether it be a smartphone, a tablet, or a personal computer.

When we talk of adding network capability to existing devices, there are a couple of options. One option is to provide a simple gateway – a network interface that

translates commands sent from networked devices into the protocol (infrared, RS-232, etc) that the standalone devices understand. The gateway solution is inexpensive to create (the gateway itself does not need to be particularly “smart”), but it presents some significant complexities for the networked devices. Specifically, each networked controller must “know” a lot about the devices it is controlling. This is particularly challenging when there is more than one networked controller, as those controllers must now share this information with one another. In the case of mobile devices (smartphones, tablets) which are sometimes on the network and sometimes powered down or away, sharing this information reliably becomes a practical impossibility.

The second option is to provide a server – a central repository of information that is constantly mediating requests form networked devices and sending out the control signals which the standalone devices require. In this situation networked clients require much less information about the stand-alone devices, so it is easier to add more clients, and to include clients of various different types. In addition, the control server provides a single version of the “state” of the standalone devices, so synchronization with multiple client controllers becomes relatively simple. Finally, the constant network presence of the control server means that it can handle and respond to events that happen even when there is no client controller present.

Because of the distinct advantages of the server approach, we designed RedEye from the ground up as a server. Thus, the same software that allows you to control RedEye from multiple iOS devices at one time has also become the foundation for things like RS-232 control and our browser-based personal computer interface.

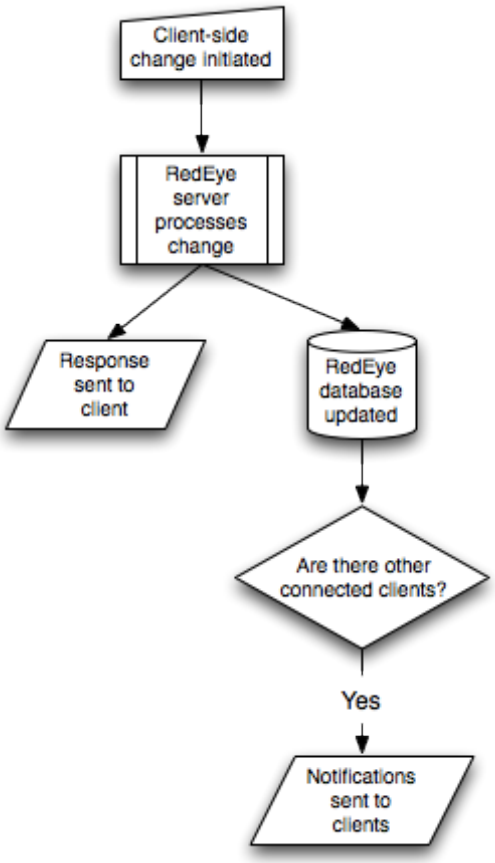
In order to take advantage of the wide array of software already available and to give us a strong foundation on which we could build future functionality, we made RedEye a full Linux server. This choice has some cost implications – running the full version of Linux (as opposed to μLinux or another embedded operating system) brings with it some relatively expensive requirements on the hardware side. On the other hand, the flexibility we gain in return has meant that RedEye can do things that were previously only possible on much more expensive systems and requiring several days of custom programming for each installation.

Relational Database

At the heart of each RedEye server is a full relational database. This database stores information about the RedEye unit itself, as well as the devices it is configured to control, and the activities used to control those devices.

Using a relational database for configuration storage gives us a couple of advantages. First, the database structure helps ensure a logical organization to the data, which makes it easy to access data in new ways and add new functionality without having to reorganize everything when we make relatively minor changes to RedEye software. Second, the database provides us with a transaction mechanism to ensure that we make meaningful changes and leave the database in a known good state if an error occurs in the middle of an update.

This last point about transactions is particularly important to RedEye's client-server design. When one client sends down a request to update RedEye's configuration, we use the transaction capability of RedEye's internal database to track the requested changes. Once those changes are complete, RedEye sends out a message to all connected client controllers notifying them of the change.



Web Interface

Another important component in the RedEye design is its web interface. Each RedEye unit runs a full web server. This not only enables RedEye's built-in browser application, it also provides a common interface that just about any networked device can access, be it a personal computer, a smartphone, or something else we haven't yet considered.

Client controllers connect to RedEye using HyperText Transfer Protocol (HTTP), and send documents in an eXtensible Markup Language (XML) format. Eventually we plan to publish an Application Programming Interface (API) for RedEye so that other programs capable of using XML and HTTP can interact

with the product, but as RedEye software is still evolving rapidly at this point we are holding off except for the most basic integration.

One bit of simple integration available today takes advantage of RedEye's built-in web server: the ability to send commands using HTTP-GET. HTTP-GET is one of the simplest and most common HTTP functions; it is the means by which standard web browsers retrieve web pages. Thus, you can invoke a RedEye command simply by pasting a URL into your browser address bar and hitting the <Enter> key. While this may not seem particularly exciting, many automation systems can invoke commands using HTTP-GET, and some of our more adventurous customers have used this feature to create their own interfaces to RedEye.

The format of a RedEye HTTP-GET command invocation looks like this:

http://[hostname]/cgi-bin/play_iph.sh?[commandpath]%20[portnumber]

where

[hostname] is the host name of the RedEye unit (either an IP address or redeye_[serialnumber].local, with [serialnumber] being the RedEye unit's serial number)
[commandpath] is the path to the command stored on the RedEye unit, and **[portnumber]** is the number of the port to which the device is attached[1]

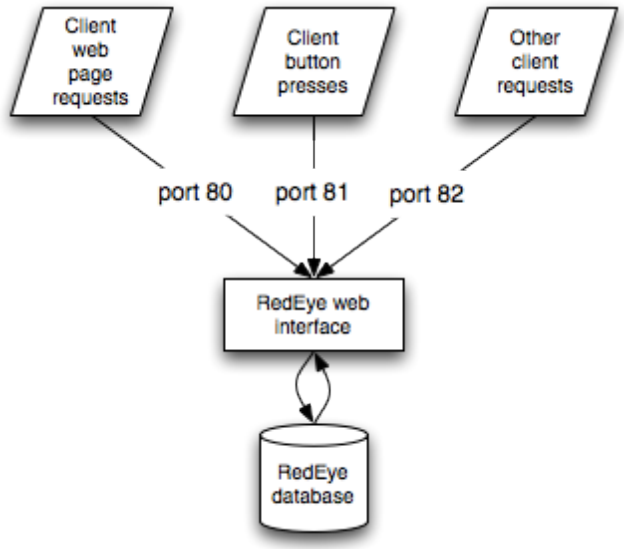
At this point you may be wondering where you are going to find the command paths in the RedEye application. There are two answers here. First, if you want to grab a copy of the URL for a single RedEye command, there is a button in the iOS which will copy the URL to your clipboard. Second, as the first method is rather cumbersome if you plan to copy a large number of commands, an alternative is to download RedEye's internal SQLite database (using the "Download Database" function from the browser application's maintenance page), and the run queries on this database to pull out the commands and paths of interest.

In fact we did not create the above capability as a means for integrating with other control systems, but rather to enable testing of infrared command files before the command itself was fully created. However, because we were using standard web technology to do this, it has evolved into an integration option which a number of customers use on a regular basis. This is the benefit of using

[1] For WiFi units, the port number will always be 1. For RedEye Pro, the port number is the infrared port number (1 to 8). Currently this feature is not enabled for RS-232 and contact closure commands.

standard technologies: generally speaking it is quite easy to interface with a number of different systems. Almost any networked device today can send HTTP-GET requests, and the vast majority are also capable of sending XML documents using HTTP-POST (which is how the main RedEye app functions).

Although we started with a standard web server, in testing we quickly found the need to optimize for performance. As a result, if you "sniff" RedEye traffic on a network today you will notice that very little information travels over the standard web server on port 80[2]. Instead, you will notice a persistent TCP connection on port 81, and transient HTTP connections on port 82. The port 81 traffic is designed for low-latency functions that need to fire rapidly – think button presses in an activity – and the port 82 traffic is for all other requests. Basically these are streamlined versions of the web server running on port 80, there to reduce overhead and memory usage so that RedEye can respond more quickly and handle more client requests. Thus, if you are configuring RedEye for remote access using port forwarding, you need to make sure you set up forwarding on three ports – 80, 81, and 82 – in order to make sure the application can work fully.



Scripting Engine

Beginning with v2.5.0 of the RedEye application, RedEye includes a scripting engine, which provides access to all of RedEye's basic functionality, plus tools that simplify the process of communicating with external devices which might use more complicated data structures. RedEye's scripting engine essentially allows you to "crack open the hood" to create your own custom commands, launch and

[2] Technically a web server can run on any port, but most publicly available web servers "listen" on port 80 (those that do not usually employ port 8080).

shutdown activities, fine tune user interface elements, monitor and respond to information coming in from external devices, and even store and retrieve data using the file system and custom databases.

Scripting is another one of these basic pillars that makes RedEye so powerful. Without a scripting interface it would be difficult to handle inbound data except in very simple ways. Limited primarily to the one-way sending of commands, RedEye would be no more than a basic remote control. With a proper scripting interface, however, you can customize RedEye to respond to various external events in meaningful ways, such as launching activities, updating client controller interfaces, and broadcasting information to client controllers.

Because this scripting engine provides much of the current customization capability in your RedEye system, most of the remainder of this manual will focus on scripting and related topics.

Part II: Scripting Basics

Introduction to Lua

One of the funny things about software developers is that they tend to have a different programming language for just about everything. Indeed, it almost seems like something developers do for fun – when you have a little free time, why not create a new programming language?

While some languages appear to exist only to satisfy their creators' desire for something that appeals to their own tastes, there are some good reasons for this variety. Perhaps the most compelling reason is that computers themselves come in many different shapes and sizes. Big enterprise servers are optimized for one kind of processing, while small, embedded microcontrollers are optimized for another, and developers have created different languages to make it easier to address these different platforms.

When we set out to add scripting capabilities to RedEye we were not interested in creating our own scripting language, but we did want to choose a language that works well with RedEye's capabilities. The Lua programming language (http://www.lua.org/) fits the bill nicely. In particular, the following traits attracted us to Lua:

1. **Lua is lightweight.** Lua compiles down to only a few hundred kilobytes, which means more of the storage space in RedEye is available for your scripts and your data, and not consumed by the scripting engine itself.
2. **Lua works well with C/C++ code.** Lua is written in C – just like RedEye – and so we were able to compile it directly into the RedEye application itself. This means that when you call system functions from your Lua scripts, often you are getting right down to "bare metal" – taking advantage of the maximum speed and efficiency that compiled code can offer.
3. **Lua is well established.** Lua is already a popular scripting language, which means that it is well tested and reliable. In addition, there are many freely available resources that make it easy to extend Lua for new purposes. We take advantage of several of these extensions in RedEye (more about that later).
4. **Lua is familiar.** Perhaps you have not programmed with Lua before. Even so, Lua's syntax shares much in common with other programming languages. It also uses a minimum of obscure symbols, instead opting for common, easily recognized keywords.

Basic Syntax

With programming languages, nothing explains better than a few examples, and so we provide those here. A quick note before we begin, though: this manual is not intended to stand as a comprehensive reference on programming in Lua, but rather just to provide a quick introduction so that we can proceed with learning how to customize RedEye. For the interest reader (or the more serious programming project) there are other learning resources available. In particular, the book *Programming in Lua* offers a solid treatment of the language and is available not only in English, but also German, Korean, Chinese, and Japanese.

First, Lua is case-sensitive. That is, it differentiates between “For” and “for”. Thus, the following are not the same:

```
print("Hello Lua")
```

versus

```
Print("Hello Lua")
```

In fact, the latter does nothing for you (well, you will get an error), whereas the former writes text to the standard output (usually the console if you are programming interactively). This brings up another point: Lua keywords are all lowercase. Since Lua is case sensitive, this helps because you do not have to use the Shift key as much.

Lua programs execute from top to bottom. Thus if you write something on the first line of a script, it will execute first. If you wish to call a function, however, you must define the function before the line of code on which it executes. Therefore the following will fail:

```
printHello()

function printHello()
    print("Hello Lua")
end
```

whereas the following is perfectly fine:

```
function printHello()
    print("Hello Lua")
end

printHello()
```

Lua doesn't care about line endings (much). Some languages let you continue a line on and on until you type an end of line character (such as a semi-colon). Others require you to type some kind of continuation character (such as an underscore) if you spill over onto a new line. Lua doesn't really care where you break your lines.

Instead, Lua pays attention to "chunks" of code. That is, it is looking for you to use correct syntax. For example, all **function** statements must finish with an **end** statement. Likewise, **if** conditions end with the word **then**, and the code they execute ends with the word **end**. Thus, the following is valid:

```
if today == "Monday" and
   thisMonth == "April" then
    expectRain()
end
```

while the following will fail:

```
if thisMonth == "December"
    expectSnow()
end
```

There are a couple of exceptions here, the biggest being the way Lua handles multi-line string literals. Specifically, if you start a string on one line using double quotation marks (") you must end it on the same line using double quotation marks. Therefore the following is *not* valid:

```
local longString = "This is a very long string of text. It has to
remain on one line if it is in double quotes like this."
```

If you want to create a multi-line string, you should surround it with double square brackets ([[and]]), as follows:

```
local longString = [[This is a very long string of text. It can
spill over to additional lines if you use double square brackets
like this.]]
```

Comments are also sensitive to line breaks. For example, two dash characters (--) denote a comment that runs to the end of the line:

```
-- The following function prints a greeting
function printHello()
    print("Hello Lua")
end
```

If you want to stretch a comment over multiple lines, or if you want to end a comment before the end of a line, you must enclose it in the aforementioned double square brackets, as follows:

```
--[[ The following function prints a greeting
    parameters: none
    return: none
]]
function printHello()
    print("Hello Lua")
end
```

Variables and Scope

Like most other scripting languages, in Lua you do not have to declare a variable before you use it, and the variable will take on a type based on the value it is assigned. For example, the following are all valid statements:

```
projectName = "Dvorak"
projectDays = 10
project2Name = "Brahms"
_projectCost2 = 512.37
project_has_started = true
local projectDescription = "Something musical"
```

As you can see, variable names can have letters, numbers, and underscore characters (although they cannot begin with a digit)[3]. They can use upper- and lower-case letters (remember, Lua is case sensitive), and there is no limitation on length.

Note that the keyword “local” preceding the last variable assignment. When you declare a variable implicitly, you are creating a global variable – one that exists during the life of the Lua environment (and since RedEye’s Lua environment is running from the moment the RedEye box boots until it shuts down, this is a long time). As a result, we generally recommend creating variables using the **local** keyword. This gives them local scope – meaning that outside the function or block of code in which they are declared they are no longer used. This allows Lua to reclaim memory from them, making your programs more efficient. It also avoids sticky situations in which a variable assignment made in one program is accidentally used somewhere else.

[3] You should also avoid variable names that begin with an underscore followed by all capital letters, as this is the format for many of Lua’s internal variables (e.g., _VERSION), which you might accidentally overwrite.

Another point of interest regarding the above variables is that we have three of the six possible Lua types represented. Specifically, we have **strings**, **numbers**, and a **Boolean** (true/false). The other types are **nil** (for undefined variables), **tables**, and **functions**.

The last two types deserve special mention. In Lua, a table is similar to an array – it can hold multiple values. In fact, tables can old other tables, so you can easily create multi-dimensional arrays. Unlike arrays, however, tables automatically size themselves, so you can easily add and remove member elements. To define a table, we use curly braces. The following creates an empty table:

```
local playlist = {}
```

Once you have defined a table, it is easy to add elements to it. One thing that makes tables really powerful is the ability "index" them using strings, numbers, etc. For example,

```
local songName = "Haitian Fight Song"
playlist[songName] = [[/MusicServer/iTunes/iTunes Music/Charles
Mingus/The Clown/1-01 Haitian Fight Song.m4a]]
```

This flexibility allows you to use Lua tables like arrays (ordered lists of elements) or like dictionaries (indexed using keys). To remove an element from a dictionary, you simply set it to nil:

```
playlist[songName] = nil
```

Functions in Lua are also quite powerful, because you can treat a function just like any other data. This means that you can assign a function to a variable, or store a list of functions in a table. The ultimate implication of this approach is that Lua is capable of object-oriented programming. Although the object-oriented approach to programming is outside the scope of this manual, we will encounter several ways in which Lua's objected-oriented capabilities make scripting much more simple and intuitive than it would be otherwise.

Operators and Keywords

We have already been dancing around some of these key bits of the language, so now it is time for a formal introduction. Lua is a streamlined language and by default has but a few operators and keywords that you need to learn. Most of the other functionality is available through extensions to the language (more on that in a minute).

As you may have picked up from the above code samples, Lua includes a handful of basic mathematical operators. Specifically, we have the following, sorted by their order of operations:

Operator	Function
^	exponentiation
not	negation (logical)
#	length (of a string or table)
-	negation (numeric)
*	multiplication
/	division
%	modulo (remainder)
..	concatenation (of strings)
+	addition
-	subtraction
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to
==	equality
~=	inequality
and	logical and
or	logical or
=	assignment

As in most languages, you can use parentheses to group operations together and thus override the default order of operations.

Another point to note is that the logical operators (and, or) are “short-circuiting” – that is, they will stop evaluating as soon as the expression becomes true. This is particularly helpful if the second half of a statement may break if the first half is false, as in the following example:

```
if (playlist[songName] ~= nil) and
   (#playlist[songName] > 0) then
    -- play the song
end
```

In the above code, if the playlist table does not contain an entry for songName, the entire condition is automatically false, and therefore Lua will not attempt to evaluate the length of the (nil) entry in the second half.

As you can tell, these operators are not terribly useful without the ability to includes them in loops, conditional statements, and functions.

Conditional Statements

Conditional statements are one of the most common programming idioms, and the syntax is similar language to language. In Lua, we format a conditional statement as follows:

```
if some_condition then
    -- do something
end
```

In the above, the **if**, **then**, and **end** keywords are required. Lua evaluates statement(s) between the **if** and **then** keywords. If they are true, then Lua proceeds to execute the statements between **then** and **end**; otherwise, Lua skips on to the next line of code after the **end** keyword.

A close cousin to this if … then logic is the if … then … else statement, which allows us to propose an alternate sequence of steps to execute if the condition is false:

```
if some_condition then
    -- do something
else
    -- do something different
end
```

We can extend this further with the if … then … elseif statement:

```
if some_condition then
    -- do something
elseif some_other_condition then
    -- do something different
else
    -- do yet something else
end
```

Conditional Loops

Conditional statements are great, but what if we need to repeat something again and again? Another common idiom is the conditional loop. It looks a lot like a conditional statement, except that it continues to execute while the condition is true.

Lua provides two types of conditional loops: the **while** loop, which evaluates its condition before each execution and loops as long as the condition is true, and the **repeat** loop, which evaluates its condition at the end of each execution and

repeats until the condition becomes true. Which flavor you choose is really a matter of programming style.

```
while some_condition do
    -- do something
end
```

```
repeat
    -- do something
until some_condition
```

Iterative Loops

Perhaps more common than conditional loops are iterative loops, which execute for a certain number of times before exiting. Lua provides two flavors of iterative loop, both using the **for** keyword.

The first iterative loop is the “numeric for” loop – a loop with a defined start and end point, and some fixed increment on each iteration. In Lua, the syntax for this kind of loop is quite concise when compared with other languages:

```
for counter = 1, 11, 2 do
    -- do something
end
```

The above loop statement does three things:

1. It initializes a variable called “counter” to the value 1
2. Executes the code inside the loop, each time adding 2 to the value of counter
3. When counter reaches the value 11, the loop executes one final time and then exits

Let’s take a look at some of the particulars. First, we do not need to include the **local** keyword before declaring “counter” – because the declaration is part of a **for** loop, Lua knows that the scope of counter should be limited only to this function.

Second, the increment value for each loop execution is 2. This number can be positive or negative. It can even have a fractional value (in Lua, all numbers are floating point, so all of the above numbers can be non-integers). In fact, we can even leave out the increment value – Lua will simply assume that we want each loop to increment by +1.

Finally, we could just as easily substitute variables for any of these numbers. For example:

```
for songCounter = 1, playlist.count do
    playSong(playlist[songCounter])
end
```

In the above loop, we use the default increment value of +1 and loop through the playlist until we have played every song. A couple of points to note here:

1. **It is common practice in Lua to begin counting at 1** (many other programming languages are 0-based). although you can do whatever you like in your own loops, there are places in the code (string parsing, for example) where you need to know this, so it is helpful to follow the convention everywhere
2. **We should never modify the value of songCounter within the loop.** Unlike many other programming languages, modifying the counter variable inside a **for** loop can lead to unexpected results.

While the above "numeric for" loops are fine, Lua also provides a much more convenient version of the iterative loop, something called the "generic for" loop – what in some languages has come to be known as a "foreach" loop. "Generic for" loops look like "numeric for" loops, except that you do not have to set up the start, stop, and increment values. The following file system example illustrates:

```
for line in io.lines() do
    -- do something with each line
end
```

What's going on here? It turns out that the file system object (io) provides a special function called an *iterator*. This iterator will keep track of all that information we had to provide in the "numeric for" loop automatically. Even better, it returns an object (in this case, a string corresponding to a single line in the file) that we can use immediately – we do not need to pull the value out of a table or anything. Pretty cool, right? Of course you can write your own iterators, but just being able to use the ones that come with existing Lua objects often is helpful enough.

One thing that can happen with these "generic for" iterators is that you may want to exit the loop early – for example, if you are searching for a particular value it does not make much sense to keep looping through once you have found that value. In this situation we can make use of the **break** keyword as follows:

```
local foundText = false
for line in io.lines() do
    if line == "text we're looking for" then
        foundText = true
        break
    end
    -- do something here
end
```

On point of interest here is that, unlike other programming languages, in Lua the **break** keyword must be the last line of code within a chunk. In the above example, it is the last line of code in the "if … then" conditional statement, so that's fine – we can go ahead and execute other code after the end of the **if** statement. The reason for this rule is that **break** immediately halts execution, and therefore any code following it is wasted.

Function Declarations

The last keyword that we need to cover is the all-important **function**. Of course you don't *have* to write functions in your Lua scripts, but if you have a chunk of code that you want to run again and again, it's much easier to write it once in a function declaration and call it over and over rather than copying and pasting it everywhere.

Lua actually provides two methods for declaring a function. The first is the most familiar, and it looks like this:

```
function playSong(filePath, repeat)
    -- do something here
end
```

Simple, no? At the top we let Lua know we are creating a function. Then we give it a name (in this case, "playSong") and some input parameters (in this case there are two: "filePath" and "repeat"). Then we write our code and finish with the **end** keyword.

The alternative method of declaring a function reminds us that in Lua functions are variables, too. Thus, we could write the above function as follows:

```
playSong = function(filePath, repeat)
    -- do something here
end
```

Normally, it does not matter which kind of function declaration you use, the second version provides some additional options when writing object-oriented Lua scripts.

Functions in Lua are extremely flexible. You can give them as many input parameters as you like, for example. You can also call them with extra parameters (the extra ones are discarded) or missing parameters (which receive the default value of **nil**). In fact, you can even give them a variable number of parameters, which may be familiar from some other languages.

Functions can also return values. What may be unexpected here is that Lua functions can actually return *more than one* value if you want, as well.

How does all this work? The following example illustrates:

```
function add(...)
    local sum = 0

    for index, value in ipairs{...} do
        sum = sum + value
    end

    return sum
end
```

1. We declare our function, giving it a name ("add") and a variable list of parameters (denoted by "…")
2. The variable parameter list is actually a Lua table, so we can iterate through it using Lua's **ipairs** function. This function actually returns two values: the index of a table member, and its value. (We only need the value, so we ignore the index.)
3. We sum the values from the table
4. We use the **return** keyword to send back the results

Just as with the **break** keyword in loops, the **return** keyword must be the last statement of a function; Lua will not execute any code after the return.

If you wanted to modify this function to return multiple values – for example, a count of the parameters passed alongside the sum – you could tweak the return statement as follows:

```
return count, sum
```

In this case, count would be a local variable which we could define as the length (remember the length operator?) of the parameter table:

```
local count = #...
```

Referencing Other Libraries

One of the most significant things you can do in any language is include functions from other code libraries. Lua allows you to include other scripts and libraries through its **require** statement.

When you include a require statement in your script, Lua will load (and run) the library you specify. Although technically you can put a require statement anywhere in your code (remember, Lua runs from top to bottom), standard practice is to place your require statements at the top of your scripts.

There is at least one Lua library that you will want to reference in your RedEye scripts: RedEye's built-in library of system functions. At the top of each of RedEye's script templates, you will find the following line of code:

```
require "systemScript"
```

What do you get for including the systemScript library? In addition to plumbing required to set up port listeners and message queues (we'll dive more into those later), RedEye's system scripts include the following functions:

Function	Use
SendMessage	Sends a message to the current RS-232 port
SendMessageToPort	Sends a message to the specified RS-232 port
Wait	Waits the specified number of milliseconds before continuing
GetVariable	Retrieves the value of a custom variable
SetVariable	Sets the value of a custom variable
GetRelayValue	Retrieves the value (open or closed) of the specified contact relay
CloseRelay	Closes the specified contact relay
OpenRelay	Opens the specified contact relay
GetSensorValue	Retrieves the value (open or closed) of the specified contact sensor
SendCommand	Transmits an infrared command
SendToggleCommand	Transmits an infrared command to achieve a particular toggle state
GetToggleStateForCommand	Retrieves the current toggle state for an infrared command
GetActivityForRoom	Retrieves the current activity for the specified room
LaunchActivity	Launches an activity for a specified room

In other words, systemScript includes just about every function you need to interact with your RedEye system.

Strings

The string data type merits particular attention because so much of programming deals with text – formatting, parsing, and presenting. From this perspective, whenever we encounter a new language it is important to understand how strings are stored and manipulated.

Lua strings are immutable – in other words, once you create a string you cannot modify it. If you concatenate one string with another, the result is a new string, not an update to the original.

To concatenate strings in Lua, we use the double dot (..) operator, as follows:

```
local firstName = "John"
local lastName = "Doe"
local fullName = lastName .. ", " .. firstName
```

The last line here takes the last name, adds a comma and a space, and then appends the first name to the end. The result is a new string, which we store in

the local variable fullName. Neither lastName nor firstName change in the process.

Simple enough, right? There is one implication of immutable strings that we need to note with caution, and that happens when concatenating large numbers of strings together. Let's take a closer look at our above example. It turns out that fullName is not the only new string we have created. There is an intermediate result – created and then immediately discarded – that may not be immediately obvious. This intermediate string variable exists because the last line of code contains two string concatenation statements. The first adds the comma and space to the last name, and the second – the one we store in fullName – adds the first name onto that result. Thus, under the covers Lua is actually doing the following:

```
local intermediateResult = lastName .. ", "
local fullName = intermediateResult .. firstName
```

On first blush this may not seem like a big deal, and in this case it really isn't. But now let's imagine that our program contains a loop, and inside this loop we are continually adding a bit more text onto the end of a string variable. Perhaps we are reading the contents of a file into memory, or maybe we are formatting an XML document as part of some communication with another device. We could easily create and destroy hundreds or thousands of these intermediate string objects. Ultimately, all of that memory allocation and string concatenation has a cost, and it can be significant.

The solution is to grab all of those bits of string that we want to concatenate, store them somewhere we can get to them, and then to perform one big concatenation at the end, with no intermediate results. Lua provides a means to do this using a special operator on the table data type – the concat function. Here's how it works:

1. We create a table to store all of the string bits we care about
2. We add our strings to the table in order
3. At the end, we call the table.concat() function on our table, which gives us a single string that we can then use as needed.

In code, here's how this looks:

```
require "table"

local stringTable = {}                          -- Step 1
local tableCounter = 1
local working = true

while working do
    local newString
    -- do some work here
    stringTable[tableCounter] = newString       -- Step 2
    tableCounter = tableCounter + 1
end

local resultString = table.concat(stringTable) -- Step 3
```

Admittedly, this requires a bit of extra effort and it is not necessary in many simple cases, but it is important to note for those times when you are doing a lot of string manipulation.

Another common need is to include special characters within a string – whether it be a newline character, or perhaps a control character that an RS-232 device requires to mark the start or end of a line. Similar to many other programming languages, Lua allows the inclusion of special characters by preceding them with a backslash (\). The following reserved characters come built into Lua:

Lua escape sequence	Result
\a	bell
\b	backspace
\f	form feed
\n	newline
\r	carriage return
\t	horizontal tab
\v	vertical tab
\\	backslash
\"	double quote
\'	single quote

In addition to the above, you can include any ASCII value by prefixing it with the backslash followed by three numbers indicating its decimal value. Thus, the start text (STX) character – ASCII value 2 – can be represented by \002, and the end text (ETX) character – ASCII value 3 – can be represented by \003.

Although Lua's basic string manipulation functions are limited to the concatenation (..) and length (#) operators, the string library includes a number of useful additions. To use the string library, you have to **require** it first:

```
require "string"
```

String Casing

It's pretty common to want to compare a couple of strings together, but sometimes the straight equality (==) comparison isn't sufficient. For example, what if you don't care about the "case" (upper or lower) of the string? Lua is case sensitive, so how do we get around this? The simplest answer is to convert everything to one case and then compare:

```
function compareNames(firstName, secondName)
    local same = false

    if string.lower(firstName) == string.lower(secondName) then
        same = true
    end

    return same
end
```

As you can see here, once we require the string library, there is a new **string** object available to us. This object defines several functions. Here we used **string.lower** to create lowercase versions of the names for comparison, but we could have used **string.upper** just as easily to compare the uppercase versions.

Converting Between ASCII and Byte Values

Particularly when programming RS-232 drivers we may need to convert from ASCII strings to byte values and back. We have already seen that we can use the backslash character to create a string literal for an ASCII value (e.g., \065 = "A"). The string library defines a function that accomplishes the same thing: **string.char**.

Perhaps more useful is the inverse function, **string.byte**. Let's say that we have some inbound text and we need to determine the starting character, which is a flag from the RS-232 device telling us which function to execute. Our processing function might look something like this:

```
function ProcessInputData(inputData)
    local operation = string.byte(inputData, 1)
    if operation == 1 then
        -- do something
    elseif operation == 2 then
        -- do something else
    else
        -- etc
    end
end
```

In the above code, we receive some data from the serial port through the local function parameter inputData. Then we use string.byte to pull out the numeric ASCII value of the first byte (don't forget – Lua is 1-based). We could have pulled the second byte by using the number 2 here, or we could have pulled the last byte by passing -1 (yes, Lua is smart enough to count backwards from the end of the string if we give it a negative number). We store this value in the local variable "operation," which we can then examine to determine what our next action should be.

Substrings

Another important part of pulling apart ("parsing") strings is being able to extract their individual pieces. Sometimes we know the absolute string length and can easily parse them based on position alone. In these situations the **string.sub** function is perfect.

In this example, let's assume that we are in the same ProcessInputData function we wrote in the last example. Let's say that if the operation charact has a value of 1, then that means the RS-232 device has set a value. The first three characters after the function byte might tell us what kind of value was set, and the second two characters tell us the actual value. In this case, we could modify our ProcessInputData function using **string.sub** as follows:

```
function ProcessInputData(inputData)
    local operation = string.byte(inputData, 1)
    if operation == 1 then
        local type = string.sub(inputData, 2, 4)
        local value = string.sub(inputData, 5, 6)
        if type == "PWR" then
            if tonumber(value) == 1 then
                Scripting.SetVariable("Power", "On")
            else
                Scripting.SetVariable("Power", "Off")
            end
        elseif type == "VOL" then
            Scripting.SetVariable("Volume", value)
        else
            -- handle other types here
        end
    elseif operation == 2 then
        -- do something else
    else
        -- etc
    end
end
```

What's going on here? We start just as we did in the earlier example by grabbing the ASCII value of the first byte. Then we read the next three characters (between index 2 and index 4 in the string) into a variable called "type" and the last two characters into a variable called "value"[4]. At this point all we have to do is examine the information in "type" and "value" and decide what to do with them.

In this example, we are storing the power state of the device in a custom variable (called "Power"). If the value is 1, then the power is on; otherwise, it is off. We are also storing the volume level in a variable called "Volume". In both cases we are using RedEye's built-in system function to store these variables.[5]

Pattern Matching

Absolute string positions are fine when strings have fixed lengths, but what can we do with something a little more fluid? Lua's string library includes a number of

[4] Given Lua's ability to search from the end of a string using negative numbers, if we assume that the string is always 6 characters long can you think of an alternative way to extract the same information into "value"?

[5] The benefit of storing these values using RedEye's custom variable facility is that we can then use this information to drive updates to the user interface on client controllers – perhaps updating a volume slider when the volume level changes, or changing a power button from red to green when system power is turned on.

powerful pattern-matching functions. While we will not cover them in depth here, a quick preview is useful.

Here's a real-world example. Some of Panasonic's Viera TVs offer an RS-232 control option. The general protocol is pretty simple: all functions begin with the STX character (ASCII 002) and end with the ETX character (ASCII 003). In addition, query responses arrive in the format of x:y, where x denotes the parameter which has changed, and y is the new value. Now, technically the first part of this response (the "x") is a fixed, three-character string, but the second half is not. Also, the colon (:) in the middle is just too tempting to pass up, because Lua makes it really easy to handle strings in this format.

Ideally what we want to do is grab the first half and put it into a variable, then grab the second half and put it into a second variable. From there we can evaluate the first half to see what changed, and then look at the second half to see how it changed. Lua's **string.match** function makes it possible to do this in essentially one line of code:

```
function ProcessInputData(inputData)
    local trimmedData = string.sub(inputData, 2, -2)
    local command, parameter = string.match(trimmedData, ".+:.+")
    if command == "QPW" then
        if tonumber(parameter) == 1 then
            Scripting.SetVariable("Power", "On")
        else
            Scripting.SetVariable("Power", "Off")
        end
    elseif command == "QAV" then
        Scripting.SetVariable("Volume", value)
    elseif command == "QAM" then
        if tonumber(parameter) == 1 then
            Scripting.SetVariable("Mute", "On")
        else
            Scripting.SetVariable("Mute", "Off")
        end
    elseif command == "QMI" then
        Scripting.SetVariable("Input", parameter)
    end
end
```

There it is – the entire inbound processor for Panasonic TVs in about 20 lines of code. Here is what is happening:

1. We start by trimming off the STX and ETX characters surrounding the input string
2. We use Lua's **string.match** function to "capture" two variables – any number of characters to the left of the colon into one variable

("command"), and any number of characters to the right of colon into the other ("parameter"). (This is where the ability for functions to return multiple values really shines.)

3. Then we have a basic set of conditional statements to evaluate the command value. Depending on the command value, we set the appropriate custom variable using information from the parameter value.

Clearly, a lot of the magic here is happening in the second argument to **string.match** – that mess of punctuation which reads ".+:.+". This is a basic pattern-matching string. Because of size concerns, Lua eschews the full regular expression library by which Perl programmers swear. Here are the patterns that Lua recognizes:

Pattern	Meaning
.	any character
%a	letters
%b	"balanced" strings
%c	control characters
%d	digits
%l	lowercase letters
%p	punctuation
%s	whitespace (including newline, tab, etc)
%u	uppercase letters
%w	alphanumeric characters (%a or %d)
%x	hexadecimal digits
%z	null character (\000)
%	escape character
[and]	character sets
+	one or more repetitions
-	zero or more repetitions – shortest match
*	zero or more repetitions – longest match
?	optional (zero or one repetitions)
^	complement character set or starting match
$	ending match

Returning back to our example, we were looking to split the input string into two based on the location of the colon. Therefore we put together a pattern expression that looked for any character (.) in some number of 1 or more (+) followed by a colon, followed by any character (.) in some number of 1 or more (+). As with most regular expressions, these are easier to put together when you know what you want than they are to read and understand on paper.

Some of the above patterns meanings are not entirely obvious. Here are a few highlights:

A **balanced** string match (%b) looks for a pair of opening and closing characters surrounding some text. This is particularly useful if you are looking for strings inside of parentheses, braces, etc. For example, the syntax to find text inside of square brackets is "%b[]".

The **escape** character (%) is a special pattern-matching escape character – it does not replace backslash (\), the standard Lua string escape character. Thus if you want a pattern that finds backslashes, you would write "\\". However, since pattern-matching adds some new special characters (e.g., $ and *), you need to escape these with % when you are searching for them. And just as you include a backslash by escaping it with itself (\\), you also include a percent sign by escaping it with itself (%%).

The **character set** delimiters ([and]) allow for the creation of custom ranges of characters. You can include a number of characters directly (as in "[abcd]"), or if your characters have sequential Unicode values, you can include them in a range separated by a dash character (thus "[abcd]" is the same as "[a-d]"). You can also use the **complement** indicator (^) to get all characters *except* those listed (i.e., "[^a-d]" will match on anything except the letters a, b, c, or d).

As we have already mentioned, the plus sign (+) requires at least one repetition. The asterisk (*) and minus sign (-) both accept any number of repetitions, but when you use * you find the longest repetition, and when you use - you find the shortest repetition. Similarly, the question mark (?) finds up to one instance, but will not match if there are more repetitions.

Finally, the carat (^) and dollar sign ($) are useful in that they help specify whether the match occurs at the beginning of a string (in the case of ^) or at the end of a string (in the case of $).[6]

Again, don't be too scared off by the regular expression syntax here – these things are usually easier to use than they are to read, and they are quite powerful, so using them is worth a couple of minutes of head scratching every so often.

[6] Actually, in our example it turns out that using a ^ at the front of the pattern ".+:.+" is a safer way to do the search. If we were to get back some text from the television which did not contain a colon, this "starting anchor" would prevent Lua from slicing the string into a bunch of small pieces and searching each one again exhaustively until it could determine that no colon existed.

Scripting Basics

That's enough of the language primer for now. What you really want to do is start writing scripts, right?

Two Kinds of Scripts

The first thing to know is that although RedEye has one scripting environment (comprised of the Lua programming language and RedEye's built-in functions as supplied by the systemScript library), there are really two kinds of scripts.

Command Scripts

One type is the **command script**. If you have used RedEye before, commands will be familiar to you. They are the individual signals used to invoke specific functions on each device you are controlling. What you may not know is that each command is stored in a file on the RedEye unit. Therefore commands have two unique features:

1. **Each command is associated to a particular device.** Also, because devices are associated to ports, we can draw a line between a command and a port.
2. **Each command is stored in a file.** Thus, commands are persistent and they have a defined location where we can access them.

Action Scripts

The other kind of script is an **action script**. You may recognize actions from their use in the RedEye app today – buttons have actions, as do activities. Often, actions point right back to commands, and, indeed, they can point to script commands as well as infrared commands. However, actions are by nature more transient then full-blown commands. Specifically:

1. **Not all actions are stored.** Sometimes a client controller will create a special action, execute it, and then throw it away. For example, when you launch an activity, the RedEye app will create an action script that contains instructions to launch the activity. This script is never stored or reused, but rather generated at the point of need.
2. **When they are stored, action scripts are stored as part of something else.** RedEye does store some actions, but always as part of a button, an activity launch sequence or activity shutdown sequence, etc. Actions

scripts do not have their own files and cannot be independently executed as commands can.

3. **Actions can change based on context.** In the same way that RedEye may create an action script on the fly, it can also add additional commands to an action script depending on what is happening in the application at a given time.
4. **Action scripts are not tied directly to devices or ports.** Because of their positioning on things like buttons and activities and their transient nature, there is no clear association between them and individual devices or ports.

Impact on Scripting

The above differences have some significant implications on the way we write scripts. Chief among these is our use of the **SendMessage** function in the systemScript library.

SendMessage is a workhorse of RS-232 communications – it allows us to transmit data over the serial port. It takes a single argument – the data (text) to transmit. On its own, this is not enough to send a message properly – we also have to know which port use for transmission. If you are writing a command script, then there is no problem here, because we know the device – and therefore the port – for the command. However, if you are writing an action script a call to **SendMessage** will fail.

In an action script, the appropriate function to use is **SendMessageToPort**, which explicitly specifies the port to which the message go. In general, however, explicitly specifying the port is undesirable because it ties your scripts to a particular configuration. If you later move your devices to other ports, your scripts will break. Therefore when creating scripts that transmit data over RS-232, we recommend creating them as commands and using the **SendMessage** function. After all, you can always execute a command as part of a button or activity launch action, so this really is not a significant limitation.

Another consideration when writing scripts is the purpose of the script and likelihood of reuse. Because you can write scripts for buttons and activity launch sequences, you could create a RedEye configuration with minimal commands and devices, and jump right into customizing your activities using scripting. While your activities would be functional, they would also be inflexible and hard to maintain. For example, you could create a play button on one activity using a custom script. What if you want to invoke the same function on a button in another activity? You could use copy and paste to duplicate your scripts, but the best solution is to create a play command once and then have each button invoke that command.

Generally speaking, our bias should be toward creating command scripts. Commands are designed for reuse, and they are visible and available throughout the application. There is nothing wrong with writing action scripts here and there, but we recommend limiting your use of them to things that are specific to a particular activity. For example, adding a button to your “Watch TV” activity that runs a script to launch the “Watch Blu-Ray” activity is a good, specific use of an action script.

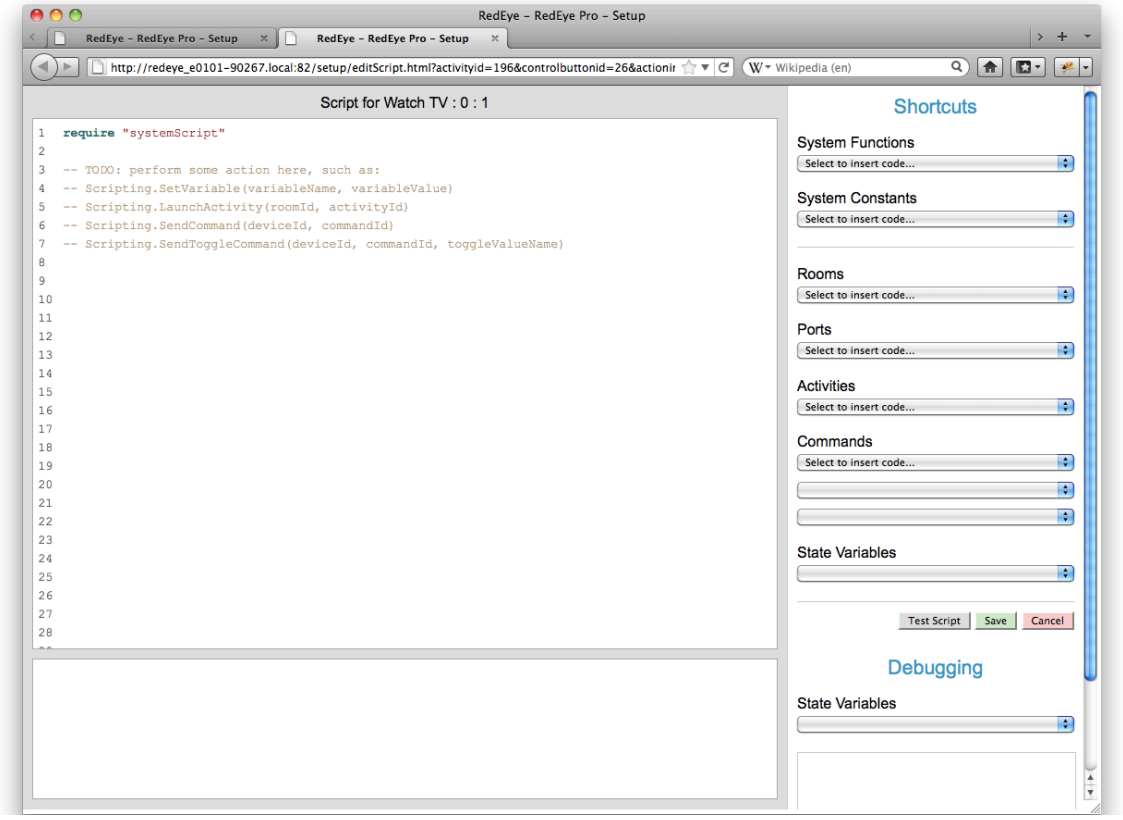
A Simple Launch Activity Script

In fact, the “Watch Blu-Ray” activity button is one of those oft-requested bits of functionality, so let’s use it as our introduction to scripting. First we need to get into our “Watch TV” activity so we can make some changes there. Scripting is a lot easier using RedEye’s built-in browser application than on a mobile device not only because you can use a regular keyboard to type your scripts, but also because a computer’s larger screen affords more space for the tools that make your scripting easier. You can edit scripts from your iPhone if you wish, but we have put a lot more effort into developing a real scripting environment through the browser app, and recommend going there whenever possible.

So, let’s launch RedEye’s browser app, switch over to the setup page, and then select the appropriate tab for our RedEye room. From there we can click on the **Activities** section and edit the button layout for our “Watch TV” activity. At the moment, this is how our activity appears onscreen:

From here let's go ahead and add the button we will use to switch to the "Watch Blu-Ray" activity. After positioning it in the upper-left corner of the layout we are ready to create the script.

First, click on the **Actions/Toggles** section in the menu to the right. Then click on the **Choose Button Action** link. From the **Type** drop-down, switch from **Command** to **Script** and then click the **Edit** button. RedEye opens a new script editing window for you to use:

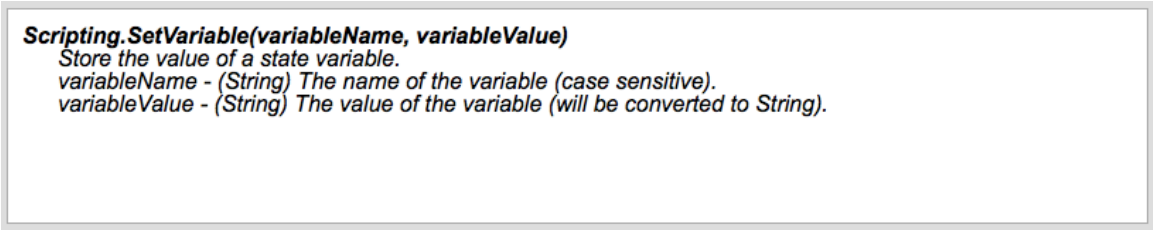


Let's take a moment to look around at the tools available to us. The script editor itself takes up most of the screen – you will notice that it is pre-populated with a template. RedEye provides different templates depending on the type of script you are editing – this is the one we get for button actions.

On the right-hand side of the screen there is a **Shortcuts** pane, which contains several quick links that insert bits of code and other useful values into your scripts when you select them. At the bottom of this pane are three buttons – **Test Script**, **Save**, and **Cancel**. If you click on the **Cancel** button, the script editor throws away any changes you have made and reloads with the last saved version of your script. Clicking **Save** stores your changes (in this case, on the button within the activity layout we are editing). You can click on **Test Script** at any time to test the script in process – there is no need to click **Save** first, nor will running a test overwrite your original script in any way.

Below the **Shortcuts** pane is the **Debugging** pane. One of the tricky things about programming RedEye is that you cannot look into the hardware to inspect what is going on. One useful debugging trick is to store information in state variables. As these are modified during your testing, you can retrieve their current values in the window below. We'll look more into debugging in a minute.

The last area in the editing window is the **Help** pane, which appears just below the editor. When your cursor is on a line of script that contains a system function or constant, this pane will display useful information and sometimes hyperlinks.


Scripting.SetVariable(variableName, variableValue)
Store the value of a state variable.
variableName - (String) The name of the variable (case sensitive).
variableValue - (String) The value of the variable (will be converted to String).

Let's go back to the editor for a minute. As you can tell, the editor will highlight text as you type to bring out Lua keywords, comments, literals, and even RedEye system functions. In fact, the editor will even help you indent your code and otherwise keep your scripts looking neat and tidy.

The first line of the template is

```
require "systemScript"
```

If you plan to invoke any of RedEye's system functions, you will need to include this statement. As a general rule, this line appears at the top of every script you write.

Below you will see a few lines of comments. For now let's go ahead and delete them – it is easy enough to add what we want using the **Shortcuts** pane. For this button we are interested in launching an activity, so let's take a look at the **System Functions** drop-down over in **Shortcuts**. Third from the bottom in that menu you will see the function "Scripting.LaunchActivity" – go ahead and select it from the list. In the editor on the left you should see the function inserted in your script (if you don't, make sure you have first clicked in the editor to position the cursor there, and then select from the drop-down menu).

At the bottom of the window in the **Help** pane you should see some information about this function. As the function name implies, it launches an activity. We need to tell it which room and which activity. We can grab those from the **Shortcuts** pane again, but first let's select the text inside the function parentheses. Why? When we select the activity out of the **Shortcuts** pane, the activity information will be pasted into the editor wherever our cursor is. If we first select the "roomId, activityId" text inside the function call, then we won't have to move around parentheses or otherwise reformat the script later.

```
Scripting.LaunchActivity(roomId, activityId)
```

After inserting the activity, there appears to be a lot going on inside those parentheses:

```
Scripting.LaunchActivity(-1--[[RedEye Pro]],
                         196--[[Activity: Watch TV]])
```

In fact, all RedEye needs to run the LaunchActivity function is a room ID (a number), and an activity ID (another number), so the following would work just fine:

```
Scripting.LaunchActivity(-1, 196)
```

The issue is that room IDs and activity IDs don't exactly come to mind easily (indeed, where have you ever seen them in the RedEye app before now?), and so while this form is compact, it is not particularly helpful. Instead, RedEye inserts a short comment after each ID so that you know the name of the room, activity, device, command, etc.

Debugging the Launch Activity Script

At this point you are probably dying to hit that **Test Script** button and see if this stuff really works. Before you do, though, let's add one more line to our script so that we can learn a bit about debugging. After the LaunchActivity line, hit <Enter> and then add the system function SetVariable:

```
Scripting.SetVariable(variableName, variableValue)
```

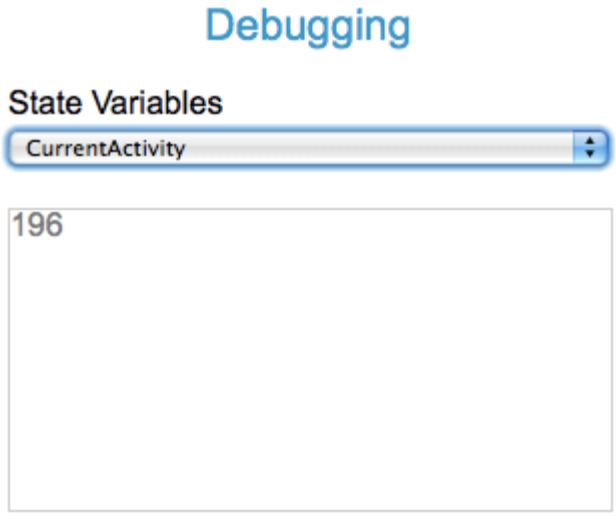
For variable name, insert the name "CurrentActivity" (within quotation marks). Then for the variable value, let's insert another function: GetActivityForRoom. This one takes an argument – room ID – so go ahead and fill that in with the room ID matching the room in which you just launched your activity:

```
Scripting.SetVariable("CurrentActivity",
    Scripting.GetActivityForRoom(-1--[[RedEye Pro]]))
```

By now you are probably getting the hang of the **Shortcuts** pane, but what exactly are we doing with this code? The SetVariable function is going to set a variable (CurrentActivity) to the value returned by GetActivityForRoom. You probably don't have a variable called CurrentActivity, but that's OK – RedEye will create one for you when you set its value for the first time. If everything goes right, then that value will be the activity ID we specified in the LaunchActivity function call on the line before.

OK – now go ahead and click the **Test Script** button.

You may notice that it takes several seconds to run this script. Why? The script execution itself is quite fast, but we have to remember what we are doing here – launching an activity. Depending on the state of your system, this could involve turning on a few pieces of equipment and waiting through delays that last up to a several seconds each before changing inputs, etc. When you are finished however, your “Watch Blu-Ray” activity should be running.



Now let’s take a gander down at the **Debugging** pane. In the **State Variables** drop-down you should have an entry for CurrentActivity, which, when selected, should display the ID for your “Watch Blu-Ray” activity.

Congratulations – you have completed your first script. Don’t forget to click the **Save** button before closing the editing window. You might also want to set the button name (“Watch BD” fits well), and icon (“Text Only” seems appropriate here), and then save your activity layout.

The PortListener Framework

Maybe you have written a couple of scripts at this point, and now you are ready to start controlling serial devices and responding to sensor inputs and that kind of thing. If so, then you'll want to take a minute or two to understand RedEye's PortListener framework.

RedEye allows you to handle inbound data and communication over serial ports using control scripts. By default, RedEye does not create any control scripts – you need to create them when you configure devices on each port. However, once you have created a control script for a port, RedEye will launch that control script whenever it boots up, and subsequently whenever you make changes to the control script itself.

Control scripts are a little different from other scripts. They need to do a couple of things:

1. **Control scripts need to be persistent**. Most scripts do a few things and then exit. Control scripts need to hang around and wait for new information to come in from the outside.
2. **Control scripts must meter traffic on the port**. RedEye is a multi-client control system. That means any number of people can be interacting with it at one time. However, devices on the other end need to have one clear picture of what to do. We cannot mix signals, sending part of a command from one place interspersed with parts of other commands from elsewhere. All of those client connections need to funnel down into one, single file list of marching orders.

RedEye's PortListener framework solves these two problems with a minimum amount of effort required on your part. Where possible we have provided a generic solution that works for all kinds of ports and devices, which allows you to focus on what makes your system unique. As a result, there is a fair amount going on "under the covers" and it is helpful to get at least a basic understanding of this so that you can make things work the way you need.

Behind the Curtain

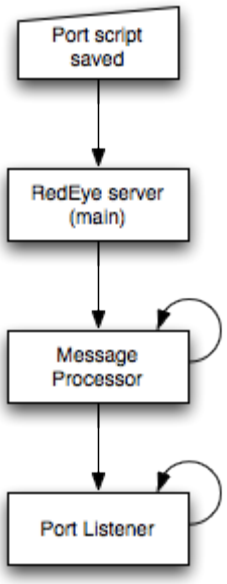
What happens when you create a port control script? First RedEye saves the control script in a file. Although you cannot test or execute a control script on its own, these scripts are not transient like action scripts, but instead are stored permanently and independently like control scripts. Next, RedEye updates the port configuration table to point to your new control script. This allows RedEye to

kick off your control script the server next boots. From there it shuts down any currently executing script for the port and fires up a new set of processes.

Although your port control script is a single file, your change actually affects three separate processes:

1. **RedEye server.** The RedEye server is the process which handles requests from client controllers and generally governs the mainline RedEye functionality on your hardware. It determines when to start the other port control processes, and keeps track of which processes are active, shutting them down as necessary (on reboot or when a control script changes).
2. **Message processor.** When you save a port control script, RedEye server does not launch that script directly. Instead, it launches an intermediate message processing script, which is the process responsible for making sure that messages sent from various client controllers to the port are handled one at a time, on a first-in-first-out (FIFO) basis.
3. **Port listener.** Once the message processor is ready to go, it launches the port listener script that you have customized. All port listeners derive from a generic port listener which takes care of all the input/output (I/O) operations. All you need to do is provide a function or two to handle the processing of inbound data, and port listener base class[7] takes care of the rest.

[7] If you aren't familiar with object oriented programming (OOP) or the concept of a base class, don't worry – you don't really have to know too much about it to use it. Suffice it to say, Lua allows us to writes script which "inherit" functionality from other scripts. Your individual port listener scripts make use of some generic blocks of code that take care of opening connections and listening on serial or sensor ports. When new data comes in, they invoke certain functions that you specify in your script. Then when your functions are called, you can deal with the data and processing you care about without having to worry about the mechanics of managing the low-level data flow.



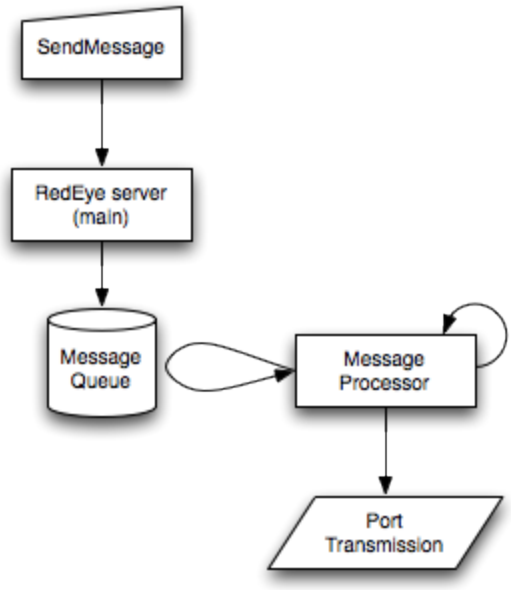
It is important to note that the code which makes both the message processor and port listener persistent is already written for you – all you need to do is handle data as it comes down the pipe. Likewise, when it is time to shut down or restart, the message processor and port listener base class take care of that work, as well. Not only do you write fewer lines of code, but you also avoid having to change your code if the internals of the RedEye server change.

Message Queuing

Now we know how everything starts up, but how do these pieces function minute to minute? There are two flows to consider here, the first being the route of an outbound message being transmitted from the RedEye server to a device you want to control.

In most cases, these messages originate as commands invoked by a client controller (although they could certainly be commands triggered by some inbound data, but we'll consider that in a moment). Whenever you call the SendMessage or SendMessageToPort functions, the main RedEye server process takes your message and writes it out to a message queue for the destination port. Each port has its own, independent message queue, and each queue works FIFO.

The message processor for the port in question is looking for changes in the message queue, and goes into action when a new message is available. It reads the message from the queue and sends the data out to the serial port. Then it processes the next message in the queue or waits until there is a new message available to process.



There is an important footnote here: the message processor only starts when you create the port control script. Therefore even if you do not plan to process inbound data on the port, you must create a port listener and save it before you can send any outbound messages. If you only add a serial port device and immediately begin writing command scripts, you will find that your commands do not have any effect (i.e., there is no message queue to receive them, or message processor to send them) until you first save the control script.

Inbound Data Processing

The second flow in the port control process is the handling of data coming into each port from external (controlled) devices. This is where your port listener-derived control script comes into play. The port listener base class opens a connection to read data coming into the port. When it receives a chunk of data, it passes this data directly to the function(s) you have defined in your script.

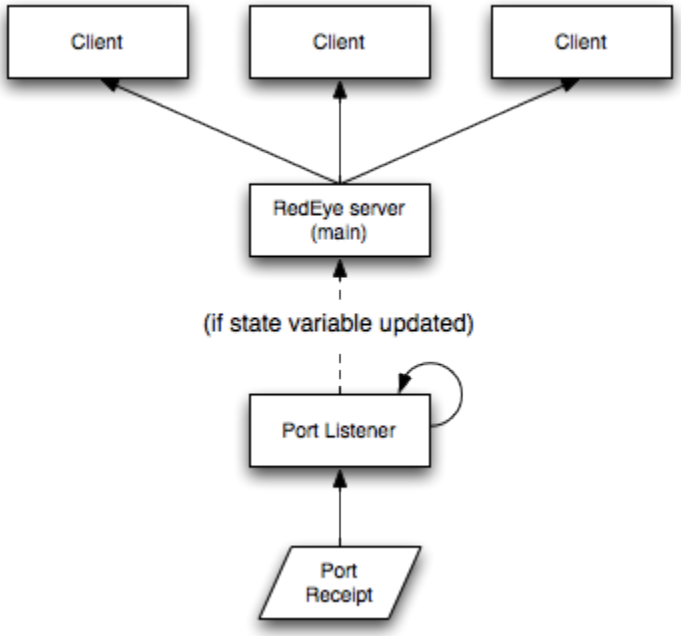
Once you have this data, it is up to you what to do with it – you can store it in a database or the file system, for example. However, one common need is to push notifications out to client controllers. For example, if you are controlling an A/V receiver and someone turns the volume knob on the receiver, you may want to update the clients to let them know that the volume level has changed. RedEye's facility for sending these updates lies in custom state variables.

Internally, RedEye uses state variables to keep track of key information about your RedEye system. For example, we store the ID of the current activity for a given room in a state variable, and we keep track of the toggle state of toggling infrared commands in state variables. When these variables change, client

controllers need to know, and so RedEye automatically sends a notification to them.

RedEye also allows you to create your own, custom state variables. These are handled in the same way as system state variables – that is, they are stored in the same place and the notification facility is the same. The difference is that you have complete control over them – unlike system state variables, you can add, delete, or change them at will.

With RedEye's state variable notifications going out automatically, if you want to update client controllers and their user interfaces, all you need to do is update the value of your state variable(s). If necessary, you can even update more than one state variable at a time. RedEye wraps your inbound data processing inside a transaction, so all changes you make within an update cycle are batched together and then broadcasted at once. This reduces the amount of network traffic and also frees you from having to worry that clients might receive one bit of information about an update but be missing another critical piece for some period of time.



Customizing Your PortListener

All the above may sound great, but how hard is it to write your own port listener? The final answer depends greatly on how complicated the protocol is for the device you are controlling, but generally speaking you can have a basic port listener up and running in just a few minutes.

Although there is a single PortListener base class which works for all types of ports (sensor, serial, etc), in practice you will use port listener a little differently in each case. Sensor port listeners are the simplest example, so let's start there.

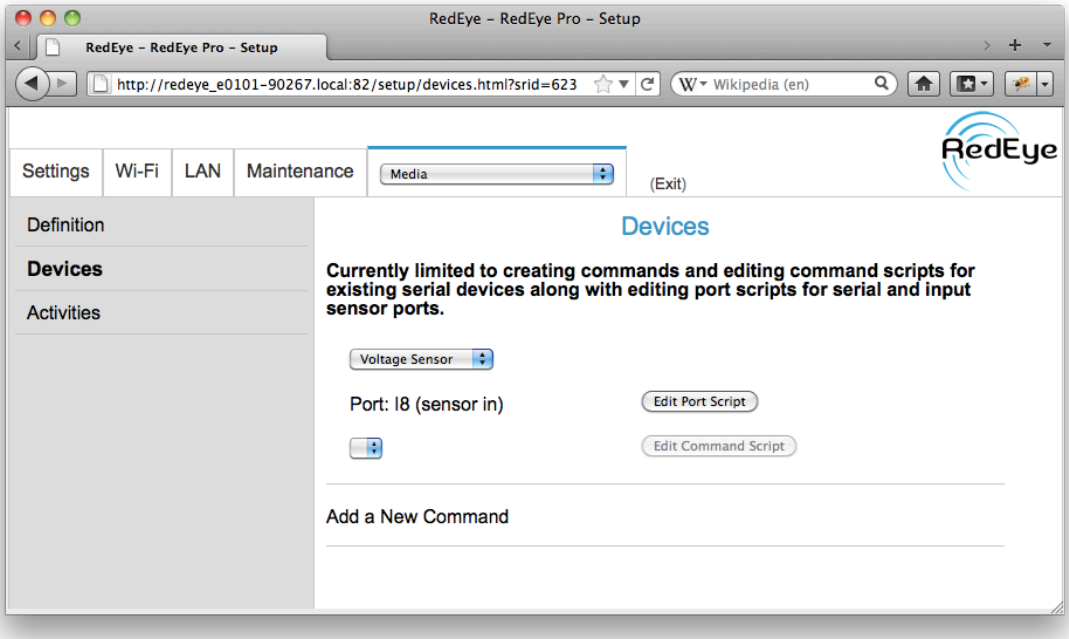
A Sensor PortListener

One of the "problems" that people cite for touchscreen remote controls is a lack of "hard" buttons – physical buttons on the controller that you can operate without having to look down or find a place on the screen.[8] With RedEye Pro's sensor and RS-232 communications, you can break free from pure touchscreen control and use many other devices – light switches, strike plates, infrared sensors – to invoke commands. For purposes of illustration, let us assume that we have a voltage sensor wired to one of RedEye Pro's contact closure sensor ports and monitoring whether the TV in our setup is on or off. With this voltage sensor in place, it would be fantastic if we could detect when someone hits the power button on the TV and respond by launching the "Watch TV" activity when the TV is turned on, or shutting down all activities in the room when the TV is turned off. Thankfully, we can wire this up with just a couple of lines of code in a port listener script.

Let's assume that we have already plugged our voltage sensor into port I8, created a voltage sensor device on our RedEye Pro, and configured the port for sensor input.[9] RedEye's browser app is our script editing vehicle of choice, so let's fire up a browser, flip over to the setup page for our Pro, and click on the Devices section. Here we can select the voltage sensor device:

[8] Having to look down at the screen to invoke a command is indeed an issue for touchscreen controllers, but hard buttons are not the only possible solution. One feature which RedEye has offered from the beginning is the ability to use multi-touch or motion gestures as button shortcuts. Both of these options provide the ability to "press" a button without having to look down at the screen. Also, when using RedEye's browser application with a PC, you can assign shortcuts to just about any key on the keyboard, giving you many more hard button options than most dedicated remote controls.

[9] When you change the mode of RedEye Pro's dual-purpose ports to sensor input, RedEye Pro puts 5V on the line to power the sensor. It is important that you do not have an infrared emitter cable plugged into the port at this time, because leaving a constant voltage on the emitter could burn out the LED inside.



When you click on the **Edit port script** button, RedEye opens a new script editing window and presents you with a template for a sensor port listener. Here we pause to note that there is no port control script running on this port yet – RedEye has only served up a template for us, and the script will not "go live" until we click the **Save** button.

At this point we could save the template and we would have a fully operational sensor port listener on I8 – it just wouldn't do anything other than listen. We'll provide functionality in a minute, but first let's take a look at what the template gives us.

At the top of the script you will notice that we have a new **require** statement in additional to the usual systemScript:

```
require "systemScript"
require "portListener"
```

Remember that RedEye provides you with a PortListener base class which gives you the basic functionality required by all port listeners. You could ignore this base class and write your own, completely custom port listener, but why go to the trouble? Instead, let's understand how the base PortListener works and what you have to do to use it.

Right after the **require** statements, there are a couple of functions: SensorOpened() and SensorClosed(). This is the place where we will put in our

custom code in a minute. For the moment we'll move on to the latter part of the script to see what is happening there.

The next line is perhaps the least intuitive part of the whole script, but it is absolutely vital to the proper function of our port listener:

```
local portId = arg[1]
```

It's clear enough that we are creating a local variable to store the port ID for our particular port, but what is "arg[1]" and where does it come from? From the bracket notation, we can assume that arg is a Lua table. We can also assume that it must be a global variable, otherwise how else would we be able to access it here?

Remember our earlier discussion about the launch process for message processors and port listeners? When the RedEye server application launches the message processor, it uses a library called Lua Task. In the task creation process, we can specify launch arguments to be passed to the spawned task, and these are available through the global variable "arg." When RedEye server launches the message processor, it populates the arg table with two bits of information, the first being the port ID, and the second being the path to our port listener script. Then when the message processor launches our port listener script, it populates the arg table with a single piece of information: the port ID. Thus in our script we have access to the port ID through the global variable arg.

It turns out that the port ID and our two custom functions are all that we need to make everything work. Thus, in the next line we create an instance of PortListener that we will use:

```
local listener = PortListener:new()
```

Then we tell it which functions to call when the sensors open and close. (This is where it convenient to be able to treat functions like data.)

```
listener:SetSensorOpenedCallback(SensorOpened)
listener:SetSensorClosedCallback(SensorClosed)
```

As you can see, we could rename our SensorOpened() and SensorClosed() functions whatever we want – we just have to make sure we pass the right name into SetSensorOpenedCallback() and SetSensorClosedCallback().

Now we can set the port ID:

```
listener:SetPortId(portId)
```

Everything is set up and ready to go. All we have to do is turn it on, which we do by telling the PortListener to listen:

```
listener:Listen()
```

So, that's it – we have everything here to make our sensor operational. But we still haven't put in any of the custom code to respond to the TV turning on and off. Let's go back up to the SensorOpened() and SensorClosed() functions at the top of the script. In the future we can ignore everything else, because this is where the real action happens.

Let's start with SensorOpened(), which is the function that the listener calls when the TV is powered down. If the TV is off, then we want to shutdown any running activity in the room. We can do this with a single call the the LaunchActivity system function:

```
Scripting.LaunchActivity(623--[[Media Room]], -1)
```

The first argument in LaunchActivity is the room ID, which in this case is 623 for the Media room. The second argument is the activity ID. Since we want to shut everything down, we pass -1 here. This single line of code is all that we need to have everything shutdown automatically when the TV turns off.

SensorClosed() is the function that the listener calls when someone turns on the TV. Of course it's possible that the TV could be turning on because we launched an activity from within the RedEye app (e.g,. "Watch DVR" or "Watch Movie"). We don't want to immediately switch to a different activity, so we'll put a check in here to make sure that there was no activity running first:

```
if (Scripting.GetActivityForRoom(623--[[Media Room]]) == -1) then
    Scripting.LaunchActivity(623--[[Media Room]],
                             624--[[Watch TV]])
end
```

That's all there is to it. With the addition of a voltage sensor and a couple of lines of code, RedEye Pro is now much more than a smartphone remote system – it is now smartly integrated with the hard power button on the front of the TV. You can imagine how this type of customization could extend to things like light switches and dimmer controls if you were to wire in a lighting system.

A Serial PortListener

Sensors are pretty easy to integrate – they are either on or off – but what about RS-232 devices? The PortListener base class for serial devices is the same, with

just a minor initialization difference. What matters here is the code that we write to handle the inbound data.

Back in our introduction to the Lua programming language, we used an RS-232-controlled Panasonic TV to demonstrate some of Lua's string manipulation functions. Let's go back to that example now and look at the rest of the port listener code.

As was the case when we added the voltage sensor and edited its port script, RedEye gives us a port listener template. This one looks very similar to the last, with a couple of key differences. At the top of the page we have the same **require** statements, because we are drawing on the same scripting libraries:

require "systemScript"
require "portListener"

After that we have a place where we can enter our custom code. Rather than a pair of functions for SensorOpened() and SensorClosed(), however, we have a single function called ProcessInputData(). Again, we'll pass on this for a minute and come back.

Next we have the now-familiar portId assignment and instantiation of the PortListener instance:

```
local portId = arg[1]
local listener = PortListener:new()
```

Then, rather than setting the sensor callback functions, we set the single data receive callback function:

```
listener:SetBytesAvailableCallback(ProcessInputData)
```

And, once again we set the port ID and tell the listener to start waiting for input data:

```
listener:SetPortId(portId)
listener:Listen()
```

In the serial port listener template there are a couple of lines of code which follow this function call. These are completely optional, but often useful: when we first connect to the RS-232 device, we are not sure of its current state. Maybe it was powered down before, but the RedEye itself has been offline for a few minutes, during which time it was powered on – the point is, if we have lost connectivity for even a few seconds, we could be out of sync. As a result, we may want to run a couple of start-up queries to find out where things are.

Running queries means sending some requests down the serial pipe to the device on the other and then receiving information back. We're in the port listener script, so receiving information is no problem. How should we send data outbound?

The answer is that by the time the port listener is up and running, the message processor has also started, which means we can send messages or commands, as necessary. Just to be safe, we should give the message processor a chance to get its queues initialized. Waiting for a second should be plenty:

```
-- make sure the message processor is ready to go
Scripting.Wait(1000)
```

After this, we can fire off a number of requests. There is no need to wait between each one, because the message processor will queue them up appropriately for us. Then the TV will respond and we will get a callback in our ProcessInputData function.

```
Scripting.SendMessageToPort("QPW", portId)
Scripting.SendMessageToPort("QAV", portId)
Scripting.SendMessageToPort("QAM", portId)
Scripting.SendMessageToPort("QMI", portId)
```

Now all we have to do is implement the ProcessInputData() method. We figured that one out before, but here it is again for reference. The inputData parameter we receive in the function call is a string, so most of this is just string parsing. And to notify clients of the changes, we also include some calls to SetVariable().

```
function ProcessInputData(inputData)
    local trimmedData = string.sub(inputData, 2, -2)
    local command, parameter = string.match(trimmedData, ".+:.+")
    if command == "QPW" then
        if tonumber(parameter) == 1 then
            Scripting.SetVariable("Power", "On")
        else
            Scripting.SetVariable("Power", "Off")
        end
    elseif command == "QAV" then
        Scripting.SetVariable("Volume", value)
    elseif command == "QAM" then
        if tonumber(parameter) == 1 then
            Scripting.SetVariable("Mute", "On")
        else
            Scripting.SetVariable("Mute", "Off")
        end
    elseif command == "QMI" then
        Scripting.SetVariable("Input", parameter)
    end
end
```

Command Scripts

Once you have your port listener up and running, you can write (and test) some command scripts. Remember: if you don't edit and save a port script, there will be no message processor running on that port, and therefore any commands that you attempt to send will have not effect.

If you've followed along through the mechanics of the PortListener framework, then writing a command script will seem relatively simple. (If you are jumping in at this point and haven't read about PortListener, then you should at least know that you need to save a port script before you can test your commands, as mentioned above.)

Most commands consist of a single call to the SendMessage() function. As covered earlier, we can avoid using SendMessageToPort() because the port is always known when we are executing a command:

```
Scripting.SendMessage("tray:open")
```

Sometimes you need to do a little pre- or post-processing around the message you send out. For example, a imagine an RS-232 Blu-Ray player that defines three commands for controlling the disc tray:

1. tray:open opens the tray
2. tray:close closes the tray
3. tray:status queries the open/closed status of the tray

This is very specific, which is exactly what we want for an RS-232 protocol. However, what if we want a single "Eject" command? The thing about the familiar Eject function is that if the tray is open, Eject closes it, and vice versa. Therefore our command script is a bit more involved (but not much). The trick here is knowing whether the tray is open or closed. For this we need to tweak the port listener script, too. Let's start there.

Let's assume that the disc player responds to the tray:status query with tray:0 if the tray is closed, and tray:1 if it is open. In the initialization section of our port listener, we would fire off a query to find out the current status of the tray when RedEye starts up:

```
Scripting.SendMessageToPort("tray:status", portId)
```

Then in our ProcessInputData() function we store the response to this query in a state variable[10]:

```
if command = "tray" then
    if tonumber(parameter) == 1 then
        Scripting.SetVariable("Tray", "Open")
    else
        Scripting.SetVariable("Tray", "Closed")
    end
end
```

This should take care of keeping track of the tray state. Now we can write our Eject command:

```
require "systemScript"

local trayState = Scripting.GetVariable("Tray")
if trayState == "Open" then
    Scripting.SendMessage("tray:close")
else
    Scripting.SendMessage("tray:open")
end
```

Presumably when you send one of the “tray:” commands the disc player will respond with an update on the tray status, but if it doesn’t, we have a couple of options.

1. We could add a post-processing step to the command. Again, we have a couple of choices here. We could set the state variable based on whether we have just opened or closed the tray, or we could send the “tray:status” query again.
2. We could add a pre-processing step to query the tray status. This is the safest choice, but we’ll have to give the disc player a chance to respond, which introduces a bit of delay. Here’s how that script would work:

[10] If you don’t see a need to notify clients of this change – and in the case of tray status, we may not – then you could store the value somewhere else. Since we have not yet reviewed other storage options yet, we’ll stick with a state variable for now.

```
require "systemScript"

Scripting.SendMessage("tray:status")
Scripting.Wait(200) -- give the disc player a chance to respond

local trayState = Scripting.GetVariable("Tray")
if trayState == "Open" then
    Scripting.SendMessage("tray:close")
else
    Scripting.SendMessage("tray:open")
end
```

Obviously the ideal situation is one in which the disc player keeps us updated when the tray opens and closes, but not all RS-232 devices have ideal protocols.

Other Scripting Tools

In addition to the built-in system functions, RedEye provides a number of other scripting tools that you can use. Some of these tools could merit an entire manual unto themselves, and most are only really necessary when you are doing some major programming, so we will leave the details for you to explore. Still, a quick introduction to what is available is in order.

File System Access

Lua's Input/Output (I/O) library is available for your use, and RedEye gives you access to a sizeable chunk of the file system, as well as some device ports. Lua.org provides an online tutorial on how the built-in I/O library works:

http://lua-users.org/wiki/IoLibraryTutorial

As far as access goes, all of your Lua scripts are available in RedEye's /devicedata partition under the /lua subfolder. When you access the I/O library the /devicedata/lua folder becomes your root folder. This means two things:

1. You are limited to this folder and its subdirectories.
2. If you want to access "/devicedata/lua/tmp," the correct path is "/tmp".

Proper Use of the Flash File System

Although you can write freely to the file system, it is important to understand a bit about writing to flash memory and the long-term implications of frequent writes. RedEye stores its programs and data on a NAND flash chip rather than use a mechanical disk. NAND flash has several advantages – it is fast, it does not fail mechanically, it is compact and consumes little power, etc.

NAND flash also has some tradeoffs. For example, it has a relatively high percentage of "bad blocks" – unusable chunks of memory. A brand new chip may have up to 1-2% of memory in an unusable state, and these bad blacks grow over time. In addition, each time a block of memory is overwritten, there is a very slight chance that it will become unusable.

RedEye uses a file system designed to work with the deficiencies of NAND flash,[11] so you don't have to worry about your data going away. However, there is

[11] Specifically, we use the Journaling Flash File System, v2 (JFFS2). If you are curious, you can learn more about JFFS2 here: http://sourceware.org/jffs2/

one unavoidable implication of using NAND flash, which is that over time the amount of available storage space slowly decreases as more and more blocks become unusable.

The rate at which storage space shrinks is directly related to the frequency of database writes. Within the RedEye server application we take care to avoid writing to the flash chip too often, and as a general practice, you should too.

All this is not to say that you should be afraid to store information in flash. It is there to use, and we can write millions of times without a noticeable impact on storage. Still, we should be careful that we do not write to flash more often than necessary.

One option that we use often in the RedEye server application which may work for some of your scripts is to store data in the temporary file system. When you write out to the /tmp directory, you are not actually writing to the flash memory. Instead, everything stored in /tmp is kept in memory. You can access it just as with any other file, but when RedEye reboots the information there is cleared out. That is usually fine for things such as the state of RS-232 devices, because under normal circumstances you need to retrieve new state information after boot anyway.

One example of RedEye's use of the /tmp file system is the message queuing system we covered in our discussion of the PortListener framework. RedEye creates its message queues as files in the /tmp/queues directory. When the system reboots, these are automatically cleared, but that is acceptable because we would need to clear out these queues on reboot anyway. Also, since we write to these queues frequently it is particularly important that we save the flash memory from this constant activity.

Accessing Devices Directly

Although RedEye provides a convenient and managed way for you to access serial ports using the message queuing system, your ability to access the file system using Lua's I/O library means that you can directly read from or write to I/O devices in your own scripts. Should you find a need to do so, RedEye currently gives you access to the following devices:

1. **/dev/ttyS1**. This is the S1 serial port.
2. **/dev/ttyS2**. This is the S2 serial port.
3. **/dev/null**. The "null" device provides a black hole where anything you write is consumed immediately. May be useful for testing in some situations.

4. **/dev/zero**. The “zero” device always returns zeros when read from, which can be useful if you need to zero out some memory or fill a stream with empty data for some period of time.

Multithreading

Although we have not discussed it directly, in out treatment of the PortListener framework we mentioned that RedEye uses Lua’s Task library to create messages processors and port listeners. We referred to these as processes, but technically these POSIX threads and not full system processes. Whatever the case, at times you may find it convenient to be able to spin off a separate thread to do some work while your main script thread executes. RedEye gives you this option through the Lua Task library:

http://luatask.luaforge.net/

XML Processing

The use of XML in data formatting is becoming increasingly more common as Internet technologies begin to replace older, proprietary formats. As we are currently working on allowing control via Internet Protocol (IP) for an upcoming RedEye release, we also anticipate the need to handle XML documents in scripts. To this end, RedEye includes the Lua XML library:

http://manoelcampos.com/files/LuaXML-0.0.0-lua5.1.tgz

SQLite Databases

File system storage is fine, but when you need something with a little more structure, SQLite is a great option. RedEye actually uses a SQLite database for its own storage, and with the LuaSQLite3 library you can do so with your own data, as well. You can read more about SQLite here:

http://www.sqlite.org/

And you can read more about LuaSQLite3 here:

http://lua.sqlite.org/

ThinkFlood, the ThinkFlood logo, RedEye, the RedEye logo, and the RedEye stylized “R” logo are trademarks or registered trademarks of ThinkFlood, Inc.

Apple, the Apple logo, iPod, and iTunes are trademarks of Apple Inc., registered in the U.S. and other countries. iPhone is a trademark of Apple Inc.

Wi-Fi, WPA, and WPA2 are trademarks or registered trademarks of the Wi-Fi Alliance.

All other trademarks are the property of their respective owners.

©2011 ThinkFlood, Inc. All rights reserved.